PowerDB-IR: Information Retrieval on Top of a Database Cluster

Torsten Grabs
Database Research Group
Inst. of Information Systems
ETH Zurich
8092 Zurich, Switzerland
grabs@inf.ethz.ch

Klemens Böhm
Database Research Group
Inst. of Information Systems
ETH Zurich
8092 Zurich, Switzerland
boehm@inf.ethz.ch

Hans-Jörg Schek
Database Research Group
Inst. of Information Systems
ETH Zurich
8092 Zurich, Switzerland
schek@inf.ethz.ch

ABSTRACT

Our current concern is a scalable infrastructure for information retrieval (IR) with up-to-date retrieval results in the presence of frequent, continuous updates. Timely processing of updates is important with novel application domains, e.g., e-commerce. We want to use off-the-self hardware and software as much as possible. These issues are challenging, given the additional requirement that the resulting system must scale well. We have built PowerDB-IR, a system that has the characteristics sought. This paper describes its design, implementation, and evaluation. PowerDB-IR is a coordination layer for a database cluster. The rationale behind a database cluster is to 'scale-out', i.e., to add further cluster nodes, whenever necessary for better performance. We build on IR-to-database mappings and service decomposition to support high-level parallelism. We follow a three-tier architecture with the database cluster as the bottom layer for storage management. The middle tier provides IR-specific processing and update services. PowerDB-IR has the following features: It allows to insert and retrieve documents concurrently, and it ensures freshness with almost no overhead. Alternative physical data organization schemes provide adequate performance for different workloads. Query processing techniques for the different data organizations efficiently integrate the ranked retrieval results from the cluster nodes. We have run extensive experiments with our prototype using commercial database systems and middleware software products. The main result is that PowerDB-IR shows surprisingly ideal scalability and low response times.

1. INTRODUCTION

Motivation. The objective of this current work is a *scalable* infrastructure for information retrieval (IR) with up-to-date retrieval results in the presence of frequent, continuous updates. IR functionality supports users who seek relevant information. IR on up-to-date data is important with novel application domains, e.g., e-commerce. Our scenario is *concurrent update* and *retrieval*, and it requires *up-to-date retrieval results* as well as *low response times*. Trying to meet these requirements is challenging with workloads or collection sizes that grow in an unpredictable way. The infrastructure must therefore be scalable. Much of its complexity is due to the requirement of 'unlimited' scalability, as we will explain.

Technology. We for our part want to use existing concepts as well as off-the-self hardware and software technology as much as possible to implement the infrastructure envisioned. Our approach is to rely on database technology to provide reliable and concurrent processing and updating of the text documents. In particular, we rely on IR-to-SQL mappings and database clusters:

IR-to-SQL mappings implement IR functionality in standard SQL for the common retrieval models, namely (extended) Boolean, vector space and probabilistic retrieval [13]. IR-to-SQL mappings are the approach of our choice: we do not need to extend the database system or to integrate an IR system.

Database clusters have been touted as a scalable infrastructure for information systems [16, 12]. A database cluster is a cluster of workstations connected by a communication network. Each node runs a DBMS, used as a storage manager with transactional properties. The idea is to *scale-out*, i.e., to add further nodes to the cluster, to meet a given performance goal. An important issue that we need to resolve is the distribution of data to cluster nodes. In this paper, we investigate several well-known physical *data organization schemes*, notably replication, partitioning, striping, and materialized views and combinations of these, and adapt them to our application scenario. This is in contrast to our previous work [9, 11] that has assumed one data organization for the cluster. Moreover, our previous studies were restricted to the Boolean retrieval model. This paper in turn covers retrieval models with ranked result sets such as vector space retrieval.

Contributions. Our approach is to combine the concepts and technologies mentioned so far. The concepts are rather general and require customizations for information retrieval, as we will explain. We have designed and implemented a prototype called *PowerDB-IR* and have investigated its characteristics in quantitative terms. This paper now makes the following contributions:

Evaluation of the database cluster. Studies on database clusters as a platform for real-world applications have focused on online-transaction processing so far [16]. Our current work investigates the characteristics of a database cluster in another important application domain, namely information retrieval. Our scenario is more general than the one mentioned just before, as we will explain.

IR functionality for different document categories. Today, many IR applications divide the document collection into categories such as 'news' or 'sports'. Existing IR systems keep statistics per category and use these to rank query results. This is not appropriate with queries over several categories since per-category statistics

.

may lead to wrong rankings. We introduce *multi-category queries* to search several document categories in order to overcome this drawback. We discuss their implementation for vector space retrieval and investigate their performance characteristics.

Data organization schemes. We adapt known data organization schemes for a database cluster to our application scenario. We use replication, striping, partitioning and materialized views and combinations of these. It turns out that different data organization schemes are appropriate for different performance requirements and different workloads. For example, striping in combination with small read-only workloads and 0.5 GB of document data reaches minimum query response times with 8 cluster nodes already.

Up-to-date retrieval results. Both updates and retrieval operate on the same data, such that query results are up-to-date. Decomposition and parallelization of requests keeps the effect of concurrent updates on retrieval performance small, as our experiments show.

Summing up, this paper is an experience report on the design, implementation and evaluation of PowerDB-IR. It says how we meet the requirements in combination, both in terms of functionality and performance. The remainder of this paper is as follows: Section 2 covers related work. Section 3 says how to implement retrieval functionality with a cluster of databases. Section 4 describes the experiments and provides a discussion. Section 5 concludes.

2. RELATED WORK

Efficient Processing of Information Retrieval Queries. Combining information retrieval with databases has already been investigated before, e.g., [2, 3, 17]. These approaches extend the infrastructure with proprietary components in order to combine IR and database functionality. Recent work has shown that response times of IR queries are competitive with commercial database systems and standard SQL for the common retrieval models [13]. But [13] has only investigated single-query workloads. [13] also does not say how to keep retrieval results up-to-date.

Concurrent Queries and Updates. Incremental index updates [4, 18] are seemingly relevant when up-to-date retrieval results are required. For instance, [4] discusses two fast incremental indexing methods and storage allocation techniques for inverted lists. Both approaches apply sophisticated disk-organization techniques that reflect the skewedness in the length of posting lists. We for our part use existing components as storage managers (e.g., RDBMSs) and do not modify them. Our philosophy is to try to reach a given performance objective by a clever combination of a (possibly large) number of such off-the-shelf components. Moreover, [4] aims at inputs in batch runs, not online updates as our approach.

PC Clusters. [1, 7, 14] investigate generic aspects of the implementation of services on a PC cluster. [16] uses a PC cluster for the TPC-C benchmark that mimics an online transaction processing scenario. An important characteristic of this benchmark is that there is a natural distribution of data and workloads: updates and queries typically go to one node only. It is relatively easy to achieve good scalability with such a restricted scenario. Our work in turn investigates a much more general scenario. Data distribution to cluster nodes is not obvious in our case and depends on the workload. Moreover, request processing typically requires data from many cluster nodes. With vector space retrieval for instance, ranking of the overall result requires efficient integration of results from the cluster nodes. This paper investigates query processing algorithms that guarantee correct integration of query results. Our previous work has not addressed this issue [9, 11].

Parallel Database Systems. Research on parallel database systems has already investigated data organization schemes that distribute the data among cluster nodes, e.g. [5]. We take over the ideas of replication, partitioning, and striping and parallelize requests at the semantic level.

3. BRINGING IR-FUNCTIONALITY TO A DATABASE CLUSTER

This section says how to enable information retrieval on a cluster of relational database systems. We use the vector space retrieval model as our running example. But our techniques apply to other common retrieval models as well.

3.1 Database Schema

In our terminology, a *document* is assigned to a *category* such as 'news' or 'sports'. The *collection* in turn is the set of all documents stored in the system.

Figure 1 illustrates the database schema that implements vector space retrieval with our approach. The figure shows tables TR_i ($i = 1, 2, 3, \ldots$) that store the documents of category i in attribute a_i. a_i stands for the name of category i. An attribute `docid` of the tables TR_i stores a system-generated unique identifier. We now proceed as follows to make documents efficiently searchable. We apply IR-specific functionality such as term extraction, stopword elimination and stemming to obtain the descriptions of the documents. In the case of vector space retrieval, we store the descriptions of the documents of TR_i in a relation IL_i with attributes `term`, `docid`, and `tf`. `term` is a descriptor of the document `docid`. `tf` (term frequency) stores the number of occurrences of the descriptor in the document. In other words, relation IL_i describes the content of a_i. Another relation T_i keeps term statistics of category i. T_i has the attributes `term` and `df`. `term` contains a descriptor per tuple, and `df` is the document frequency of the descriptor, i.e., the number of documents that contain it.

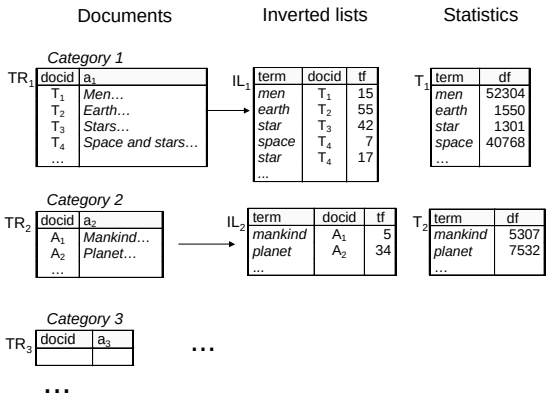

Figure 1: Database schema for IR functionality

In the following, we will discuss a service interface for information retrieval and its implementation using the above database schema. Our motivation is that the service implementation encapsulates the IR-specific functionality such as term extraction and the storage management using complex SQL statements.

3.2 Document Retrieval Service

Service Interface. Clients invoke a *retrieval service* with the following interface to search for relevant documents:

$$\{(rsv, document)\}\ RetrieveDoc(\{a_i\}, \{term\}, k, rt)$$

The input parameter is a query comprising a set of category names, a set of query terms or a query text, the maximum number of hits to

return, and a threshold on the retrieval status value of result documents. The service returns retrieval status values of the result documents and their titles, e.g., the first 50 characters of the document.

The user may specify one or several categories as the category parameter of the query. If one specifies a query with only a single category, the system evaluates the query only on the documents in that particular category using the statistics of that category. We speak of a *single-category query* in this case. Several definitions of retrieval status value for vector space retrieval are available. We use the following one where d stands for a document and q for a single-category query on category a.

$$RSV(d,q) = \sum_{t \in terms(q)} tf(t,d) \cdot idf(t)^2 \cdot tf(t,q) \qquad (1)$$

t denotes a term, and $tf(t,d)$ and $idf(t) = \log \frac{N^a}{df^a(t)}$ are its frequency in d and inverted document frequency, respectively. We say that a document d *qualifies* for a query q if the retrieval status value (*rsv* for short) of d with respect to q is above the threshold of the query, and d is among the top k matches.

So-called *multi-category queries* are appropriate in other situations: think of an online information system with job offers. A user may want to retrieve job offers for the categories 'information technology' and 'academia'. But he does not care in which of these two categories the keywords of his query occur. The problem with such a query is now that IR (except for Boolean retrieval) relies on the statistics over all categories that occur in a query. Vector space retrieval for example uses document frequencies of terms. In order to provide a correct ranking the retrieval status value with multi-category queries depends on *multi-category statistics* ($stat^c$ for short), namely $N^c = \sum_{a \in \{a_i\}} N^a$, $idf^c(t) = \log \frac{N^c}{\sum_{a \in \{a_i\}} df^a(t)}$, where N^c and N^a denote the number of documents in all queried categories and in category a, respectively. $df^a(t)$ denotes the per-category document frequency of term t with the documents of category a (see Table 1). The retrieval status value of a document d for a multi-category query q is then:

$$RSV^c(d,q) = \sum_{t \in terms(q)} tf(t,d) \cdot idf^c(t)^2 \cdot tf(t,q) \qquad (2)$$

This definition equals Definition 1 in the case of only one query category.

In the following, we will discuss a possible implementation of the vector space retrieval model, first for single-category queries, then for multi-category queries.

Service Implementation with Single-category Queries. Previous work has proposed standard SQL implementations of data access for Boolean, vector space and probabilistic retrieval models [8, 13] using the database schema from Section 3.1. We take over these implementations and encapsulate them in a single routine called **IRQuery** throughout this paper. This routine takes the name of the document category, the query text, the retrieval model, the hit threshold, and the *rsv* threshold as input parameters. There is another input parameter, namely multi-category statistics, which is optional, to be explained in the next paragraph. This parameter is omitted with single-category queries. **IRQuery** then processes the query at the database system using the statistics in T_i and returns the top k hits to the query, using the respective retrieval model.

Service Implementation with Multi-category Queries. Merely invoking **IRQuery** is not sufficient to evaluate multi-category queries, since our approach keeps statistics on a per-category basis in the term statistics table T_i for category a_i. Using single-category statistics may lead to wrong rankings with multi-category queries. In order to avoid such inconsistencies, multi-category queries in PowerDB-IR compute the correct multi-category statistics during query processing before calling **IRQuery**. Algorithm **RETRIEVEMULTI** reflects this. Its first step is to retrieve single-category statistics for each category and to use them to compute the multi-category ones. The second step executes the retrieval query in parallel at the categories. **IRQuery** takes the multi-category statistics as input parameter and retrieves the top k hits with a retrieval status value (*rsv*) larger than *rt* from each category. The third step of **RETRIEVEMULTI** integrates the results from the second step and returns the ranking. Our experiments will quantify the overhead of the integration step, i.e., Step 1.

```
Algorithm RETRIEVEMULTI
Parameters: Query q, categories A, model m, integer k, threshold rt
var hits := ∅;
begin
    // Step 1: Collect and integrate statistics
    for each category a ∈ A do in parallel
        Get per-category statistics (df^a(t), N^a); end;
    Compute multi-category statistics stat^c (idf^c for Equ. 2);
    // Step 2: Execute query for each category
    for each category a ∈ A do in parallel
        hits := hits ∪ IRQuery(a, q, m, k, rt, stat^c); end;
    // Step 3: Post-processing and output of results
    Sort hits by RSV; Return k hits ranked highest (doc and RSV);
end;
```

3.3 Document Insertion Service

Clients of PowerDB-IR invoke the following *insertion service* to insert a new document d:

$$integer\ InsertDoc(a_i, d)$$

The input parameter is a category name and the document. The return value is a status flag.

The implementation of the service is as follows: suppose that the service inserts a new document d to category a_i in relation TR_i. IR-specific routines analyze d in a first step. This includes term extraction, stopword elimination and stemming. Maintenance of the index table IL_i and the statistics is needed to keep retrieval results up-to-date. We insert the document in an independent database transaction as the following algorithm shows. Besides, we generate a database transaction that inserts the descriptor and its frequency $tf(t,d)$ into IL_i for each descriptor t that occurs $tf(t,d)$ times in d. An additional transaction manager on top of the database system guarantees atomicity of a service invocation.[1] The algorithm receives descriptor information as its third input parameter. It is a dictionary that stores the descriptors and their corresponding frequencies.

```
Algorithm INSERTDOC2DB
Parameters: category a_i, document d, dict<term,tf> T
begin
    begin transaction;
    insert into TR_i values (docid, d);
    commit transaction;
    for each term t ∈ T do in parallel
        begin transaction;
        insert into IL_i values (t, tf(t), docid);
        update T_i set df = df+1 where term = t;
        commit transaction;
    end;
end;
```

In other words, we decompose maintenance of the document, index, and statistics tables into independent database transactions that can appear in any order. In fact, the independent subtransac-

[1] See the longer version of this paper for a discussion [10].

$stat^c$	multi-category statistics	n	number of cluster nodes
n^g	number of cluster nodes in node group g	N^c	number of documents with a multi-category query
N^a	number of documents in category a	$\tilde{N}^g$	approx. number of documents in node group g
N_i^g	number of documents stored at node i of node group g	$tf(t,d)$	number of occurrences of term t in document d (term frequency)
$idf(t)$	inverted document frequency of term t (single-category query)	$idf^c(t)$	inverted document frequency of term t (multi-category query)
$df^a(t)$	document frequency of term t in category a	$df^g(t)$	document frequency of term t in node group g
$i\tilde{d}f^g(t)$	approximate inverted document frequency of term t in node group g	$df_i^g(t)$	document frequency of term t in node group g on node i

Table 1: Abbreviations

tions can also be executed in parallel. This yields a high degree of intra-service parallelism. It should lead to good speedup and scaleup characteristics in a cluster of databases.

3.4 Physical Data Organization with a Database Cluster

A database cluster allows for different physical data organization schemes. We map the document, inverted list and statistics tables of our schema to one or several cluster nodes using document categories, striping, replication, or a combination of these techniques. We now define some notions and give an overview of the data organization schemes.

Partitioning the nodes of the database cluster results in so-called *node groups*. Replication as well as storing of categories occur at the granularity of node groups: *storing categories* assigns a category and its dependent data to one node group. This means that some node group stores TR_1, IL_1, and T_1; and another one stores TR_2, IL_2, and T_2, and so on. *Replication* in turn copies a category and its dependent data to several node groups. *Striping* instead occurs inside node groups, i.e., all data assigned to a particular node group is striped over the nodes of the group.

Example 1: Figure 2 illustrates the assignment of the two categories 'information technology' with documents T_i and 'academia' with documents A_i to three node groups and a cluster of six nodes. Category 'information technology' is assigned to node groups 1

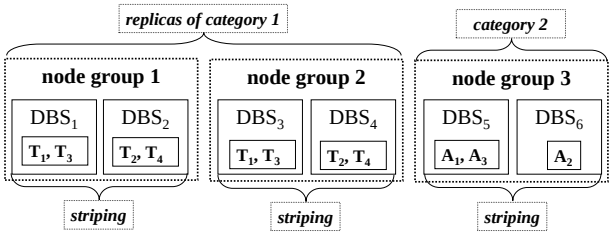

Figure 2: Organization of document data on a database cluster

and 2. Each such node group stores the document, index and statistics data of the category. In other words, category 'information technology' is replicated at node groups 1 and 2. Category 'academia' in turn is only assigned to node group 3. Data is striped inside each node group. ◇

Assigning categories to different node groups has the advantage that single-category queries go to only one node group. We expect this to yield a good distribution of workloads among categories and to increase retrieval throughput. Any copy in turn can evaluate a given query with replication. The drawback is that each copy must process a given update. The *degree of replication* is the number of copies of a category. Striping randomly distributes the data among the nodes of a node group, e.g., by hash partitioning over the `docid`. The number of these nodes is the *degree of striping* of the node group. Updates only go to one node in a striped group. A retrieval service instead must execute at all nodes of the group. Having said this, we now introduce a parameter, the so-called *striping-replication ratio* $srr = \textit{degree of striping}/\textit{degree of replication}$ and assume that all node groups have the same size. The srr then defines the actual physical data organization of the cluster. For instance, striping over two nodes with four replica yields an srr of $2/4$. The idea is to adjust the srr in order to meet the performance requirements of the application. An important objective of our evaluation is to quantify the effect of srr for different cluster sizes and workloads. In all cases, we keep document, index, and statistics data, i.e., tables TR_i, IL_i, and T_i for some i, at the same node.

Retrieval on a Database Cluster. If documents are randomly distributed inside node groups as with striping, each node holds a representative sample of the documents stored at the node group. In this case, the per-node statistics are approximations of the group-wide ones: The approximate number of documents in the node group is $\tilde{N}^g = n^g \cdot N_i^g$ with n^g number of nodes in group g and i being a node of group g. The approximate inverted document frequency $i\tilde{d}f^g(t)$ of a term t in group g is $i\tilde{d}f^g(t) = \log \frac{\tilde{N}^g}{n^g \cdot df_i^g(t)} = \log \frac{N_i^g}{df_i^g(t)}$. The approximate retrieval status value of a document d for a query q with vector space retrieval is

$$R\tilde{S}V^g(d,q) = \sum_{t \in terms(q)} tf(t,d) \cdot i\tilde{d}f^g(t)^2 \cdot tf(t,q). \qquad (3)$$

Thus, query processing does not need to fetch per-node statistics for all nodes in a group to integrate them to group-wide ones. Instead, it takes the per-node statistics as a representative sample of the group-wide ones. The following algorithm shows this in pseudo-code for a striped node group g that stores the content of category a.

```
Algorithm RETRIEVESTRIPE
Parameters: group g, Query q, model m, integer k, threshold rt
var hits := ∅;
begin
    // Step 1: Execute query at each node of group g
    for each node i ∈ g do in parallel
        // call IRQuery for node i
        hits := hits ∪ IRQuery@i(a, q, m, k, rt); end;
    // Step 2: Post-processing and output of results
    Sort hits by RSV; Return k hits ranked highest (doc and RSV);
end;
```

The first step executes the retrieval query in parallel at the nodes. It retrieves the top k hits from each node using **IRQuery**. The second step integrates the results from the first step and returns the ranking.

Insertion on a Database Cluster. Routing of insertion requests depends on the data organization with a database cluster. We ship the request only to those node groups that store the documents of the document category of the request. All copies process the inser-

tion with replication. The hash value of the document id determines the node of the document with striping.

4. EVALUATION OF POWERDB-IR

We have carried out numerous experiments to find out if PowerDB-IR indeed meets the initial requirements. We start with a description of our experimental setup in Section 4.1. Section 4.2 reports on our findings and discusses the outcome of our experiments.

4.1 Experimental Setup

Documents. The document collection in our experiments is a complete snapshot of the usenet newsgroups server at the CS Department of ETH Zurich as of October 1999. After eliminating messages with only binary content, the collection consisted of about 420,000 documents in searchable categories. The amount of data is 500 MB of ASCII text. Each of the following experiments starts with all the documents. The database size is 1.7 GB with indexes and 1 GB without. This roughly corresponds to a scaling factor of 1 for the text retrieval benchmark in [6].

Workload Patterns. A tuple (a,b) denotes the workload in our experiments, where *a* and *b* are the number of concurrent retrieval and insertion requests, respectively. *a* is the number of clients with a query currently being processed. *b* has the analogous meaning. Each client immediately submits a new request when it has received the result of the previous one. We also refer to these clients as streams. Our motivation to use streams is that the number of concurrent service requests in the system is always the same for a given workload tuple. The workload tuples of the experiments are $(0,1)$, $(1,0)$, $(1,1)$, $(5,5)$, $(10,10)$, $(20,20)$, ... , $(60,60)$. Each insertion request inserts around 400 tuples to the index tables and inserts the text.

Queries and Updates. The experiments work with streams of insertions and queries. The clients run on a workstation that was not part of the cluster. We have used another 250 MB of textual data with news messages for the insertion streams. The insertion streams in our experiments selected new documents at random from this data. We have synthetically generated queries for the query streams from the terms of the documents in a first setup, similarly to [6]. Queries return at most 20 matches, i.e., $k = 20$. A query contains 2.05 terms on average. This is the average query length reported by search engine providers [15]. Another setup investigates the effect of query length on retrieval performance. We have additionally investigated complete news message body texts as queries (roughly 350 terms or 2KB) in this setup. Queries and insertion services do term extraction and Porter stemming next to stop-word elimination.

Hardware and Software. We currently use PCs with either one 400 or one 600 MHz Pentium Processor, 128 MB of RAM, and an interface to a network with a data transmission rate of 100 Mbit/sec (switched duplex Ethernet). We have tested the system with 1, 2, 4, 8, and 16 workstations. All PCs run the Microsoft Windows 2000 Advanced Server operating system software. We use the object transaction monitor COM+ that ships with Windows 2000 as middleware product. Our home-grown PowerDB architecture organizes distributed processing of IR services on top of the database cluster [9]. The DBMS at the cluster nodes is Microsoft SQL Server 2000.

4.2 Outcome and Discussion of the Experiments

We start with a discussion of our findings from experiments with different retrieval granularities, i.e., single-category vs. multi-category queries. The following paragraphs report on the scalability of our database-cluster infrastructure when using the different physical data organization schemes for document data discussed in Section 3.4.

Single-category Queries vs. Multi-category Queries. The following experiments investigate the quantitative characteristics of different query processing granularities, i.e., single-category and multi-category queries. Figure 3 reports on experiments that compare response times of single-category and multi-category queries over 16 artificially generated categories for a cluster of 16 nodes and 16 node groups. Multi-category queries go over all 16 categories. Single-category queries in turn distribute evenly among those categories. As one would expect, single-category queries are faster than multi-category ones. However, we have not expected that the difference is so large. For instance, it is by a factor of 4 with high workloads such as $(40,40)$. We have conducted additional experiments to shed more light on this surprising finding. We want to find out what exactly the overhead of the integration step of multi-category queries is as compared to queries using per-node statistics for the same number of nodes. We investigate two setups: the first one stripes documents randomly over the nodes (4 or 8 in the experiments) and processes queries using per-node statistics. The other one processes queries using multi-category statistics, i.e., it runs multi-category queries over the same data, but using the additional integration step. Figure 4 graphs the outcome of these experiments. Processing with per-node statistics is at least 10% faster. How does this relate to the previous experiments where the difference between single-category and multi-category queries is by a factor of 4? In the previous series, single-category queries are sent to their node group, i.e., one node in the experiment, and execute there using per-node statistics. Multi-category queries in turn use multi-category statistics and have to execute on all nodes. Variances in the execution times between nodes are high (not shown in the figures), and multi-category queries have to wait for the results of all nodes. Moreover, the average query load per node is much higher with multi-category queries. This is our explanation why they take so much longer. The performance overhead of the integration step in isolation is around 10%.

As we have expected, single-category queries in the presence of many categories yield small response times even with high workloads. Multi-category queries in turn are rather expensive. Materialization of multi-category statistics of categories that frequently appear together in multi-category queries avoids the overhead of statistics integration and speeds up multi-category queries. We have not investigated this issue any further because such combinations are highly application-specific.

Scalability of Service Implementations with Striping. In order to assess the scalability of PowerDB-IR with striping and one node group we have conducted experiments with different cluster sizes for mixed workloads with concurrent insertion and vector space retrieval. Figures 5 and 6 show response times for insertion and retrieval, respectively.

Insertion response times increase nearly linearly from low to high workloads for all cluster sizes. This is what we would expect: more concurrent services compete for resources, mainly disk I/O, with increasing workloads. This leads to poor insertion response times: They reach an unbearable average of more than 100 seconds for middle workloads with one node. Having more nodes leads to significantly better response times, e.g., 25 seconds with 16 nodes and high workloads. The reason is that striping distributes the insertion load evenly among nodes.

A positive finding with retrieval is that average response times on a cluster with 16 nodes are interactive, i.e., 5 seconds or less, even with high workloads such as $(50,50)$ (cf. Figure 6). This is

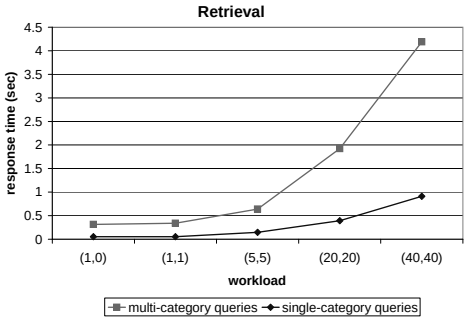

Figure 3: Single-category vs. multi-category queries

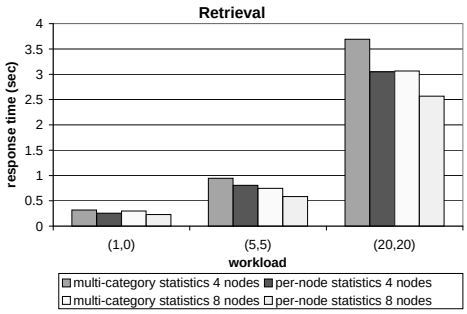

Figure 4: Query processing with multi-category and per-node statistics

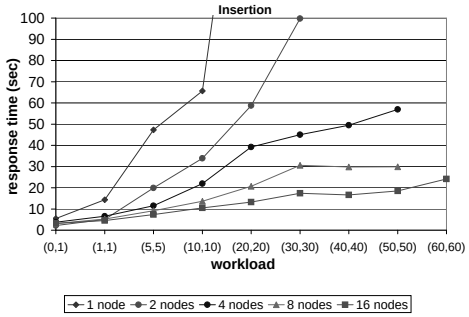

Figure 5: Response times with striping: insertion

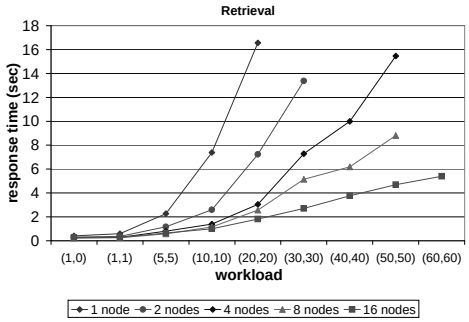

Figure 6: Response times with striping: retrieval

the case only for much smaller workloads such as $(20,20)$ with smaller cluster sizes. We were somewhat surprised by the fact that the scaleup of retrieval is less than linear, but that scalability of insertions is close-to-linear (cf. Figure 7). In other words, increasing the cluster size by a factor of 2 increases the maximum retrieval throughput taken from the range of workloads $(1,1)$ to $(60,60)$ by a factor significantly smaller than 2, e.g., 1.3 when scaling from 8 to 16 nodes. The reason is that a given query has to be executed at each cluster node with striping. The number of queries being processed at each node at a given point of time does not decrease when scaling the cluster size. On the other hand, there is a gain in throughput, for which we offer the following explanation: the amount of data that a query has to go through at a node becomes smaller with a higher degree of striping.

A positive observation in turn is that the effect of running updates concurrently to retrieval has only little effect on retrieval performance (cf. Figure 6). Going from $(1,0)$ to $(1,1)$, i.e., from retrieval only to retrieval with concurrent updates, increases retrieval retrieval response times only by 8% with 16 nodes. The same also holds for much higher workloads such as $(50,0)$ and $(50,5)$ as in Figure 11: the effect on retrieval performance is less than 20%.

We have conducted further experiments with queries comprising complete documents as query texts with around 200 distinct query terms. Retrieval response times tend to be significantly larger than with the experiments from Figure 6. Nevertheless, increase of performance by scaling out the cluster is the same as with short queries. It only takes more cluster nodes to achieve performance similar to shorter queries.

From this series of experiments with striping we have learned that query throughput does not scale linearly. A follow-up question is what we gain in terms of query response time if we increase the degree of striping. Figure 8 addresses this issue. The figure graphs average retrieval response times with the single query workload $(1,0)$ and vector space retrieval with striping. Increasing the degree of striping up to 8 nodes yields smaller response times. With 8 nodes, it is 0.23 seconds. When striping over more than 8 nodes, the overhead of distributed query evaluation is larger than the gain from striping. We conclude that query response times do not benefit from a degree of striping of more than 8. The reason for this somewhat surprising finding is that the index completely fits into main memory with 8 nodes already. A further increase of the striping degree does not compensate the additional overhead with distributed query evaluation. The speedup for high workloads (cf. Figure 6) is only due to a better distribution of the workload when scaling from 8 to 16 nodes with striping.

Replication in Combination with Striping. The previous experiments have shown that the increase in query throughput with striping is limited to a relatively small number of nodes already. A possibly better alternative is to combine striping with replication. This requires to choose a striping-replication ratio. Its effect on query response times and throughput with different workloads in a database cluster is not obvious and warrants investigation.

Using a cluster of 16 nodes, we have investigated the effect of varying *srr* from $16/1$, i.e., only striping and no replication, to $1/16$, i.e., full replication without striping. We have also looked at retrieval-intensive workloads such as $(100,0)$ and $(100,5)$. Figures 9 and 10 graph query and insertion throughput, respectively. Maximum system throughput is 50 queries per second with full replication, i.e., 16 node groups. At the same time, our system provides competitive response times of 2.5 seconds (not shown in the figure). Insertion in turn does not benefit from an increase in replication (cf. Figure 10). So far, this is what one would expect. A less obvious observation in turn is that retrieval throughput does not scale linearly when we increase the degree of replication using the same number of cluster nodes. With the retrieval-intensive workload of $(100,0)$ for instance, retrieval throughput increases by a factor of 5 when going from $16/1$ to $1/16$. It still increases by

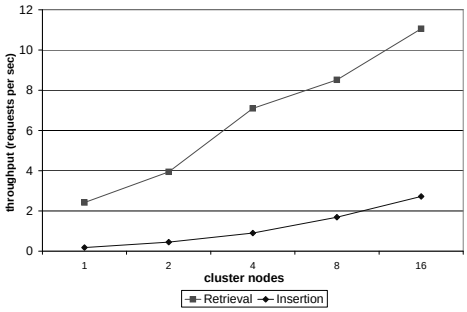

Figure 7: Scalability of maximum throughput

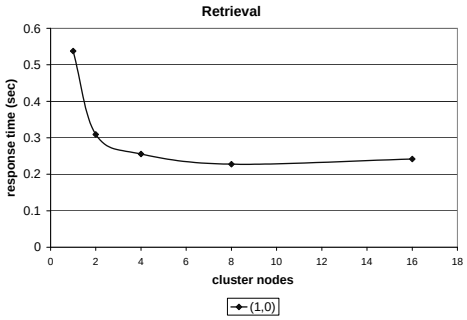

Figure 8: Retrieval response times with striping and small workloads

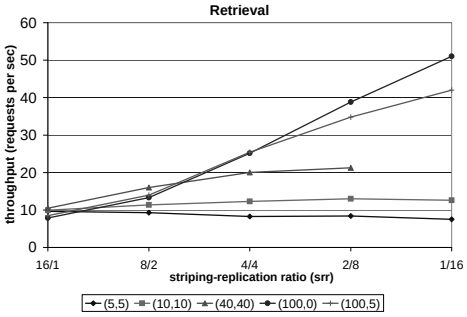

Figure 9: Combinations of replication with striping: retrieval throughput

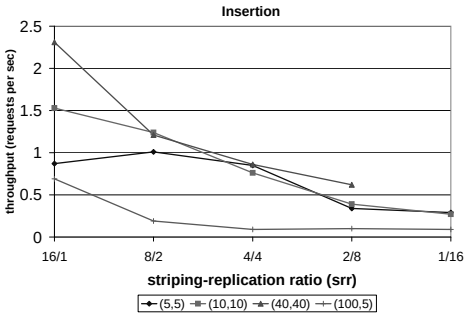

Figure 10: Combinations of replication with striping: insertion throughput

a factor of 4 when we allow for concurrent updates in the same setting. Our explanation is that minimizing *srr* gives away the advantage of striping, i.e., main memory lookup for queries. Nevertheless, the positive effect of a higher degree of replication is still larger than the disadvantage of reduced striping.

Scalability of Retrieval with Document Categories. Another dimension of the space of design alternatives is the number of categories. The respective experimental setup has the following characteristics: we have generated collections with 1, 2, 4, 8, and 16 artificially generated categories. All categories contain the same volume of document data, and we have assigned categories to separate node groups of size 1. Thus, number of categories and cluster size increase at the same rate. To study the effect of categories on single-category queries query streams with these experiments contain single-category queries only. Hence, a query goes to exactly one category, and queries distribute evenly to categories. We have measured query response times for the workloads $(1,0)$ and $(10,10)$ as well as for query-intensive ones such as $(50,0)$ and $(50,5)$. Figure 11 plots the outcome of the experiments.

Having a workload distributed among 2 or 4 different categories instead of 1 yields a high improvement in response time, as one would expect. The improvement is smaller with a further increase in the number of categories since average workloads per node are already small.

Thus, having more categories when the collection size increases leads to significant performance improvements for single-category queries. Performance improvements are in the same order of magnitude as with replication.

Scalability with Increasing Collection Size. Another important issue is the performance impact of the collection size when the number of categories is fixed. We have investigated the issue by increasing the number of cluster nodes, and increasing the collection size at the same rate. In contrast to the previous series of experiments, the number of categories is constant in this setting. We have started with the 'normal' collection size (collection size 1 or scale 1 for short) and cluster size 1. We have scaled both collection and cluster size up to 16 and have metered the effect on retrieval response times. Figure 12 reports on the outcome of these experiments. There is only a small difference in retrieval response times for increasing cluster and collection sizes with constant workload. A surprising finding is that performance actually improves when increasing collection and cluster sizes. The reason is that load balancing is easier on a larger cluster.

5. CONCLUSIONS

This paper has discussed the design, implementation and evaluation of PowerDB-IR. PowerDB-IR provides a scalable infrastructure for information retrieval with up-to-date retrieval results in the presence of frequent updates. PowerDB-IR uses a *database cluster* to achieve scalability: one adds further nodes to the cluster to achieve higher performance ('scale-out'). Our prototype relies on *service decomposition* to increase parallelism and on *IR-to-SQL* mappings. Let us look back at the requirements from the introduction. The first one was that *retrieval results must be up-to-date*. Retrieval and update services operate on the same data with PowerDB-IR, and updates are immediately available to retrieval. This study has demonstrated that the overhead of concurrent updates on retrieval performance is surprisingly small. Another important requirement was *scalability*. Our experiments have shown that PowerDB-IR indeed scales well: 'scaling-out' the cluster yields *interactive response times* even in the case of high workloads. Moreover, performance of PowerDB-IR remains constant with growing collection sizes when increasing the cluster at the same rate. We have investigated alternative *physical organization schemes for document data on a cluster of databases*. Our experi-

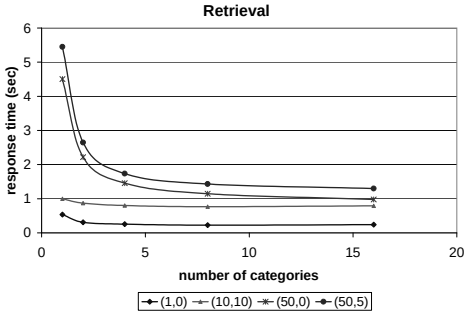

Figure 11: Increasing number of categories: response times

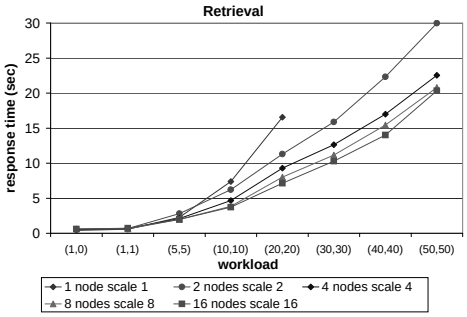

Figure 12: Increasing collection and cluster size: response times retrieval

ments show that different schemes are appropriate in different settings. System performance with small read-only workloads for instance reaches its maximum with 8 nodes already.

Our general conclusion is twofold: First, our system PowerDB-IR is a scalable infrastructure for applications that require concurrent update and up-to-date retrieval of document data. We have also shown that a database cluster is well-suited as an infrastructure for a broader range of application scenarios, compared to what has already been investigated.

Acknowledgement. We are indebted to Rory Chisholm for much help with all aspects of this study.

6. REFERENCES

[1] D. Andresen, T. Yang, and O. H. Ibarra. Toward a Scalable Distributed WWW Server on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 42(1):91–100, 1997.

[2] D. Barbará, S. Mehrotra, and P. Vallabhaneni. The Gold Text Indexing Engine. In *Proceedings of the Twelfth International Conference on Data Engineering, New Orleans, USA*, pages 172–179. IEEE Computer Society, 1996.

[3] K. Böhm, K. Aberer, E. J. Neuhold, and X. Yang. Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM. *VLDB Journal*, 6(4):296–311, 1997.

[4] E. W. Brown, J. P. Callan, and W. B. Croft. Fast Incremental Indexing for Full-Text Information Retrieval. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 192–202, 1994.

[5] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data Placement in Bubba. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, USA*, pages 99–108, 1988.

[6] S. DeFazio. *Overview of the Full-Text Document Retrieval Benchmark (In: The Benchmark Handbook – Jim Gray (ed.))*, pages 435–487. Morgan Kaufmann, 1991.

[7] A. Fox, S. G. Y. Chawathe, E. Brewer, and P. Gaulthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP'97), St. Malo, France*, pages 78–91. ACM Press, 1997.

[8] O. Frieder, A. Chowdhury, D. Grossman, and M. McCabe. On the Integration of Structured Data and Text: A Review of the SIRE Architecture. In *Proceedings of the First DELOS Network of Excellence Workshop on Information Seeking, Searching and Querying in Digital Libraries, Zurich, Switzerland, 2000*, pages 53–58. ERCIM, 2000.

[9] T. Grabs, K. Böhm, and H.-J. Schek. High-level Parallelisation in a Database Cluster: a Feasibility Study Using Document Services. In *Proceedings of the 17th International Conference on Data Engineering (ICDE2001), April 2-6, 2001 Heidelberg, Germany*. IEEE Computer Society, 2001.

[10] T. Grabs, K. Böhm, and H.-J. Schek. PowerDB-IR – Information Retrieval on Top of a Database Cluster. Technical report, Database Research Group, ETH Zurich, 2001. Available at: http://www.dbs.ethz.ch/∼grabs/papers/cikm2001long.pdf.

[11] T. Grabs, K. Böhm, and H.-J. Schek. Scalable Distributed Query and Update Service Implementations for XML Document Elements. In *11th International Workshop on Research Issues on Data Engineering: Document management for data intensive business and scientific applications (RIDE-DM'2001), 2001, Heidelberg, Germany*. IEEE Computer Society, 2001.

[12] J. Gray. How High is High Performance Transaction Processing. In *High Performance Transaction Systems Workshop, Asilomar, USA*, 1999. Available at: http://research.microsoft.com/∼gray/hpts99/talks/-Gray_Jim.ppt.

[13] D. A. Grossman, O. Frieder, D. O. Holmes, and D. C. Roberts. Integrating Structured Data and Text: A Relational Approach. *Journal of the American Society for Information Science (JASIS)*, 48(2):122–132, Feb. 1997.

[14] Inktomi Corp. The Inktomi Technology behind HotBot. Technical report, Inktomi Corp., 1996.

[15] S. Kirsch. Infoseek's Experiences Searching the Internet. *SIGIR Forum*, 32(2):3–7, 1998.

[16] Microsoft Corp. Building High-Performance Databases Using Microsoft SQL Server 2000 Federated Database Servers. Technical report, Microsoft Corp., 2000.

[17] H.-J. Schek and P. Pistor. Data Structures for an Integrated Data Base Management and Information Retrieval System. In *Eigth International Conference on Very Large Data Bases, Mexico City, Mexico, Proceedings*, pages 197–207. Morgan Kaufmann, 1982.

[18] A. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental Updates of Inverted Lists for Text Document Retrieval. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, USA, 1994*, pages 289–300, 1994.